ARL-TR-7449 ● SEP 2015

US Army Research Laboratory

Wearable Notification via Dissemination Service in a Pervasive Computing Environment

by Somiya Metu, Laurel Sadler, and Robert Winkler

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-7449 ● SEP 2015

US Army Research Laboratory

Wearable Notification via Dissemination Service in a Pervasive Computing Environment

by Somiya Metu, Laurel Sadler, and Robert Winkler
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
Sep 2015	Final	01/2015–08/2015

4. TITLE AND SUBTITLE
Wearable Notification via Dissemination Service in a Pervasive Computing Environment

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Somiya Metu, Laurel Sadler, and Robert Winkler

5d. PROJECT NUMBER
R.0013626.6.163.3

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-CII-B
2800 Powder Mill Road
Adelphi, MD 20783-1138

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-7449

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes an architecture of wearable sensors in the context of an Army tactical environment. The architecture is implemented in functional software that integrates the sensor data, performs predictions to determine contextual gesture information, and disseminates this information to other Soldiers and computing assets where the presentation is properly adapted. This prototype framework demonstrates the feasibility of how a heterogeneous pervasive computing environment can integrate context, state, and environment in a manner that would be transparent to a Soldier's common operations.

15. SUBJECT TERMS
pervasive computing, Android-based wearable application

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Somiya Metu
Unclassified	Unclassified	Unclassified	UU	26	19b. TELEPHONE NUMBER (Include area code) 301-394-1398

Contents

List of Figures

1. Introduction

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it," (Weiser 1991). Mark Weiser's view of ubiquitous computing in 1991 was seen as visionary at that time and it is only recently that the promise of Weiser's vision is actually being realized. Weiser makes an analogy comparing electricity to computing. He explains that hundreds of volts coursing through wires in walls may have been intimidating at one time, but is now accepted as commonplace without any fear or consideration. Weiser (1991) argued that similarly, hundreds of computers will come to be equally invisible and accepted, simply being used unconsciously to accomplish everyday tasks. The National Institute of Standards and Technology (NIST) defines pervasive computing as the emerging trend toward numerous, casually accessible, often invisible computing devices, that are frequently mobile or imbedded in the environment and connected to an increasingly ubiquitous network structure.

Whether it be labeled as a tactical, ubiquitous, or pervasive computing environment (PCE), networking, computers, and sensors abound all around humans' daily activities. The environment that Weiser envisioned and NIST defined is a contemporary reality in the civilian, non-governmental agency, and military domains. However, despite the availability of the infrastructure, technologies that transparently exploit pervasive computing assets are not tightly interconnected or fully utilized to meet humans' dynamic information needs. The question of what technological architectures and integrations are necessary to deliver such a capability in tactical environments remains an open and active research question. Key issues such as how might commercial hardware be used to implement methods to contextualize information with environmental and physiological state and how this information might be disseminated in dynamic tactical environments still need to be addressed.

This report describes an architecture of wearable sensors such as armbands, optical glasses, and intelligent time devices in the context of an Army tactical environment. The proposed architecture is implemented in functional software that integrates the sensor data, performs predictions to determine contextual gesture information, and disseminates this information to other Soldiers and computing assets where the presentation is properly adapted given the receivers' state/status. This prototype framework is demonstrates the feasibility of how a heterogeneous PCE can integrate context, state, and environment in a manner that would be transparent to a Soldier's common operations.

The report is organized as follows. Section 2 provides background on PCEs. Section 3 presents the system design architecture and discusses reasoning for implementation concerns. Section 4 covers implementation details on specific software methods to realize the capability and discusses the software operation. The final section concludes with a summarization of the research and work.

2. Background

Evolving from distributed systems and mobile computing, pervasive computing builds on the research of these 2 areas and introduces additional interesting research concepts: smart spaces, invisibility, localized scalability, and uneven conditioning. Figure 1 describes the logical relationship of the research issues in distributed systems, mobile computing, and pervasive computing.

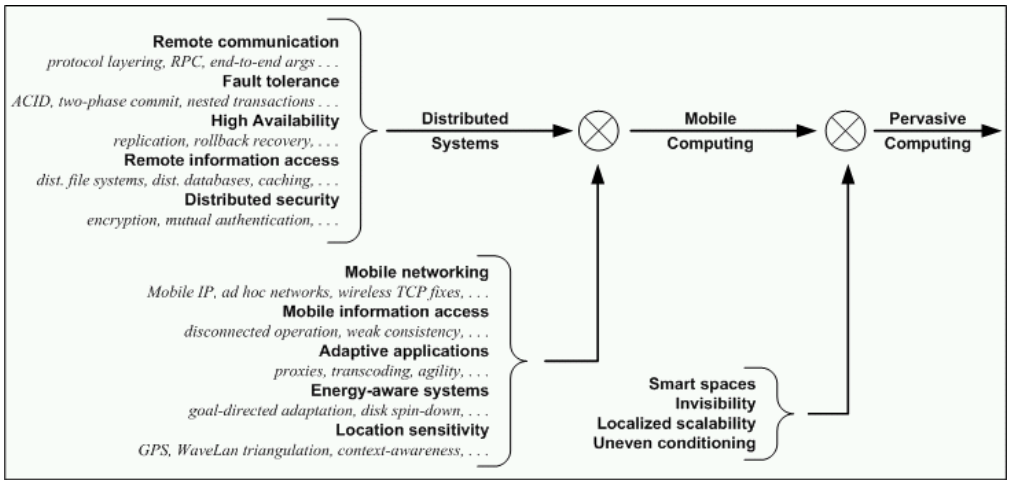

Fig. 1 Taxonomy of pervasive computing systems research (adapted from Satyanarayanan [2001])

Figure 1 describes the current PCE research issues, as defined by Satyanarayanan. It also differentiates the research issues for pervasive computing as compared to mobile and distributed computing. It is important to note that although Fig. 1 labels the research issues for each topic discretely, the research issues in each topic build on one another, i.e., pervasive computing has all of the issues identified for mobile computing in addition to the items under its label. The relevant issues related to this report are invisibility and localized scalability.

The idea expressed by Weiser (1991) as a definition of invisibility is the complete disappearance of pervasive computing technology from a user's consciousness. In practice today, it would be difficult to achieve this definition. However, if a PCE continuously meets user expectations, rarely presents surprises, and at the same

time delivers a modicum of anticipation, the subconscious interaction described by Weiser can be approximated (Satyanarayanan 2001). Invisibility stems from Weiser's original vision where the interaction with the computing technology is below the user's active consciousness. In other words, the technology interacts with the user in a manner that is not disruptive and naturally supports users' activities. The definition of localized scalability proposed by Satyanarayanan focuses on the intensity of interactions between a user's personal computing space and the surroundings. Specifically, localized scalability deals with the effective management of information exchange between users and their surroundings (Plymale 2005). Critical to the issue of localized scalability is the utility of the data delivered to applications in a PCE.

Consider the following example. A squad of Soldiers and small tactical robots are moving along a path in typical point formation, where Soldiers and robots on the edge of the squad are out of line of sight of the point man. An unattended ground sensor triggers an enemy-ahead alert transmitted via the sensor network to the point Soldier. The point Soldier receives the alert and not wishing to alert the enemy, executes a "rally-on-me" (come to where I am – Fig. 2) hand signal. The wearable device on the Soldier recognizes the gesture and notifies the rest of the squad (robots and humans), who may not be able to visually see the gesture, via the inter-squad communications network.

Fig. 2 Rally-On-Me: Come to where I am hand signal

The concept of invisibility, includes several mobile networking issues that include application adaptability and context awareness. The above example illustrates the

invisible or active nature of pervasive computing, shown in the gesture recognition, passive alerting, and non-visual tipping. The interactions with the tactical information and computing devices is natural and does not require directed interaction. Similarly as an illustration of localized scalability only the information that is necessary is transmitted dynamically adapting both content and transport mechanisms appropriately for the context.

It is difficult to characterize PCE systems without some discussion of context. To be effective, PCE-based systems require context-aware components (Want et al. 1995). PCE applications need to have some idea of user context shifts, i.e., changes in the user's position, history, workflow, or resource interests. If the PCE is described as a 2-component system, one component that is the user and a second that is an information resource, both of which may be changing context, then it is critical that applications in this environment be able to adapt (Narendra et al. 2005; Satyanarayanan 2001). Context exists in both components, as ubiquitous computing and connectivity are of little value without ubiquitous data (Cherniack et al. 2001).

The issues of invisibility and context awareness are not uniquely discrete because context awareness can affect invisibility. If an inappropriate amount of data is delivered to a device that is incapable of using it, invisibility is lost. If a user attempts to access an information resource that is no longer available and the system is incapable of responding to the request, invisibility is lost. In the literature, these types of problems are typically classified under context awareness, specifically presence awareness.

The following sections outline a software development approach for designing a prototype tactical PCE system that could implement capabilities illustrated in the above example. Specific technical details are provided to document the development approach.

3. Design Architecture

Figure 3 illustrates the overall design architecture and communications between components. The design consists of 3 handheld devices, a wearable armband sensor (Myo armband), an optical wearable (Google Glass) device or smartwatch, and intelligent communication software to facilitate information dissemination and retrieval. The Myo armband is a device to detect hand gestures. For example, the Myo armband provides the kinetic data needed to detect an Army-specific gestures that might be performed by a Soldier. This gesture information once identified, can then be communicated to other Soldiers (represented by the other handheld devices) via a wired or wireless network and displayed on either Google

Glass or the smartwatch using a local network transport (e.g., Bluetooth) with the individual Soldier's handheld. The custom communications application Dissemination Service (DisService) is used by each handheld to communicate the gesture information with the other handhelds.

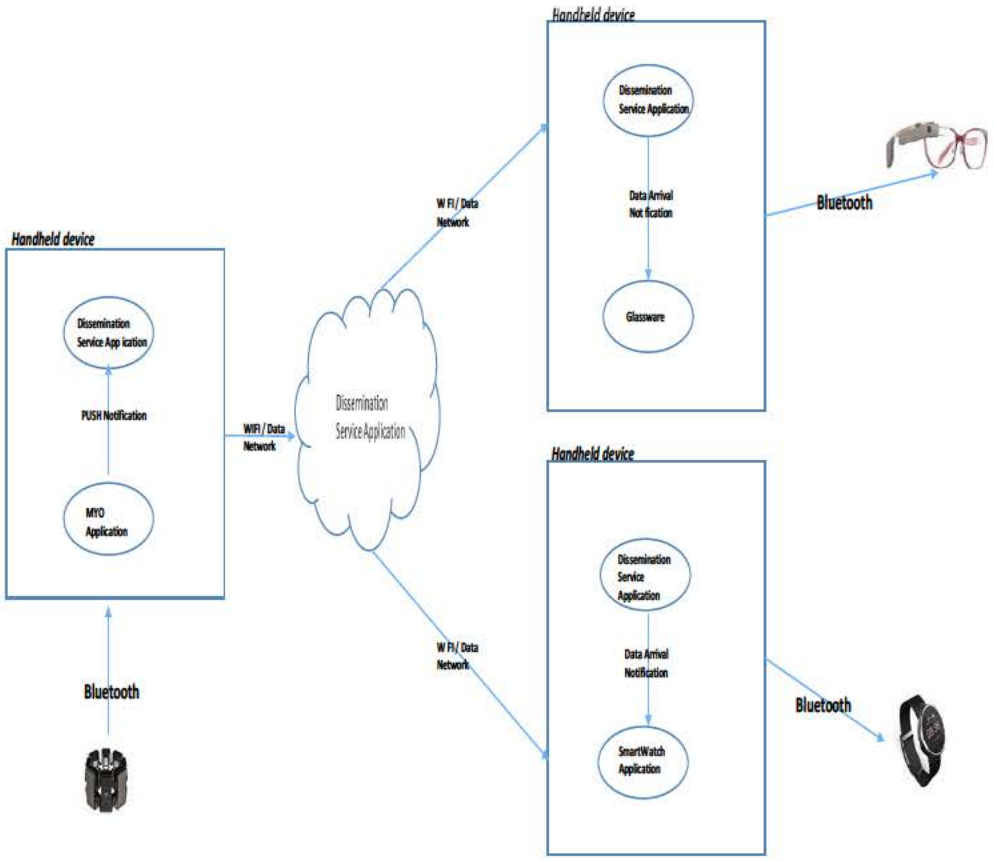

Fig. 3 Architecture for wearable exploitation in a PCE

The wearable devices communicate to the local handheld devices via Bluetooth[1] and the handhelds communicate with each other using the DisService application. Each wearable has a dedicated handheld that hosts DisService. DisService is used for information dissemination between the handhelds outside Bluetooth range. Once the information reaches the handheld, it is passed on to the paired wearable via Bluetooth. This way information can be exchanged between wearables outside Bluetooth range. Bluetooth technology has been selected for the communication link between the handheld and the wearables. In addition to the fact that most wearable devices already use Bluetooth, it also has the advantage of a small power requirement, which is crucial for a dismounted Warfighter. The notification messages from the handheld to the wearable are intermittent and require only

modest bandwidth thus making Bluetooth technology as an appropriate choice. As depicted in Fig. 3, we have use 2 different types of wearable devices, Smart Glass and a smartwatch, as presentation devices. This would later enable us to analyze suitability of a notification media type on a particular presentation device. For example, an image animation or a short video could be targeted to Glass as opposed to a smartwatch for convenient viewing on the Glass's screen just above the line of sight.

All of the software developed uses Google's Android open-source software stack intended for mobile devices such as cell phones and tablets. The target platform Android 4.4 (API 19) was used in this effort, and the Galaxy Nexus phone was the handheld used for testing the implementation. The Android Wear platform is the Google application programming interface (API) for smartwatches, Google Glass, and other wearable devices. The target platform Android 4.4W (API 20) was used for developing Wear applications. Google Glass XE22 and Moto 360 smartwatch was used for testing purposes. Android Studio 0.8.6 has been used as an integrated development environment (IDE) for software development. This IDE has Android and Android Wear support for development of Android-based applications. Our selection criteria for which devices we used in the system were that the device had to be open and we preferred the device if it supported Android Wear API so we could rapidly integrate new devices and share code between them.

As denoted in Fig. 3, DisService provides the means of communication from handheld to handheld/handhelds or, in essence, Soldier to Soldier/Soldiers/ unattended sensors/robots/intelligent systems. This wearable application uses a custom DisService for dissemination of the data message. DisService is a peer-to-peer, disruption tolerant, message passing, communications software-level protocol. It provides capabilities that address challenges to resiliency for disconnected, intermittent, and limited (DIL) networks such as those that exist in tactical PCEs. DisService supports store and forward delivery of data and caches data wherever possible in the network, thereby making it disruption tolerant and improving availability of data.

The point-to-multipoint feature of DisService is of particular relevance to this wearable application where a Soldier wearing the armband needs to send a command signal to multiple recipients. DisService accomplishes this task through subscription management and "pushing" data within a "group." For example, a client will subscribe to a particular "group" of interest. Each subscription may also have an associated priority, request sequenced data, and reliable delivery of messages. If reliability is requested, missing messages are retransmitted. Groups are used to organize the information being disseminated and may also be tagged

to differentiate between multiple types of data. Using DisService, an application that disseminates information can either push data or publish data to the group. When information is pushed, it is delivered to all applications that have subscribed to the corresponding group. Whereas "published" information only pushes metadata describing the information and allows the client nodes to request the full information as necessary.

In the architecture shown in Fig. 3, the push protocol is applicable. Within the functionality of this architecture DisService not only allows the client to receive all information of interest, but also filters/reduces extraneous data received by the client. Pushed information messages can also have expirations in order to avoid a new subscriber from receiving an irrelevant command signal. In this manner, DisService allows the clients (other wearable devices) to subscribe to the same "group" that the armband uses to "push" the command signal. Thus, disseminating the command signal to multiple recipients in 1 push addresses issues of localized scalability by using less bandwidth, adding network tolerance and reliability, and expanding the range of the transmission. Moreover, it does so while being theoretically transparent (invisible) to the users and sensors that exist in the tactical PCE.

4. Framework Implementation

A gesture from the Myo wearable is propagated to other wearable devices/presentation devices using several Android-based applications. These applications subscribe to the same group in DisService, which facilitates information exchange between the applications. Figure 4 depicts the gesture propagation. The Myo application is responsible for connecting to a desired Myo wearable. It collects and formats the raw data from the Myo wearable while a gesture is being performed. The accumulated data are then passed to a naïve Bayes model integrated with the Myo application for gesture classification. The identified gesture is then pushed to DisService. The Glass application and the Smartwatch application receive the identified gesture as a result of subscribing to the same group in DisService as the Myo application. These applications choose an appropriate media-type to display the received gesture in the presentation devices. The detailed implementation of Android-based applications that facilitates gesture propagation is discussed below.

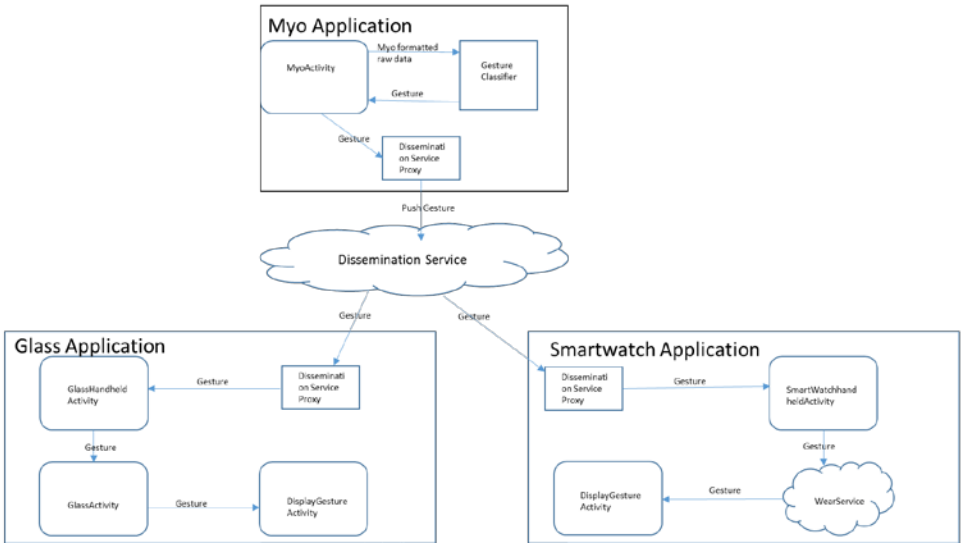

Fig. 4 Gesture propagation between wearables

4.1 Myo Application

The Myo armband from Thalmic Labs is a gesture-control armband. It has 8 muscle-sensing modules that can be strapped on the forearm to detect hand gestures. The Myo Android software development kit (SDK)[2] is included with the Myo armband. The development kit has been used to augment an existing Android-based Myo-enabled application, MyoApplication, from Thalmic Labs. The application contains a main activity, *MyoActivity*, which has been extended to implement the DisService interface. The *OnCreate* method is invoked where the activity, *MyoActivity*, is initialized. In this method, *setContentView* method is invoked to facilitate placement of user interface (UI) elements in *MyoActivity*. Following this, the *hub* class from the Myo SDK is instantiated and initialized using the MyoApplication identifier. The hub provides access to 1 or more Myo instances that may be available. After *hub* initialization, the *hub* instance is registered for any *DeviceListener* callbacks. This would enable the Myo application to receive events from the Myo devices. The events include changes in orientation, pose, Myo device lock/unlock, etc.

The Myo Android SDK comes with a built-in activity for scanning and connecting to a Myo device. This activity, *ScanActivity*, gets launched from the main *MyoActivity* to scan for any nearby available Myo devices for connection. The available Myo devices are listed for the user. Once an available Myo device is selected, it attempts to establish a connection between the handheld and the selected Myo device via Bluetooth. Following the connection of the armband to

the handheld, an asynchronous task, *initDisServiceTask*, is executed. In this task, a proxy to DisService is instantiated and initialized. The proxy subscribes to the wearable group in DisService. MyoApplication is then registered as a listener to the subscribed proxy. This would enable MyoApplication to send and receive notifications from other wearables via DisService.

MyoActivity overrides the *OnPose* and *OnOrientationData* methods, respectively, from the Myo API that get called when a new pose is identified or a change in the orientation is detected by the Myo armband. *MyoActivity* provides the appropriate interface to enable the user to initiate or end pose and orientation detection. The data accumulated in the process are passed on to a classifier for an appropriate gesture detection by *MyoActivity*. Once the application acquires a gesture string from the classifier, it attempts to pass the gesture string to other wearables. An asynchronous task, *pushMessage,* is executed. In this task, the reference to the initialized DisService proxy is used to send the gesture string to DisService. The push method of the proxy instance is invoked, which takes the gesture string as 1 of its parameters.

4.2 Glass Application

Google Glass is a wearable technology developed by Google. The Glass Development Kit (GDK) is an add-on to the Android SDK that can be used to develop glassware that can run directly on the Glass. Using the GDK and Android SDK, a glassware that runs on the Glass wearable and an Android-based application that runs on the handheld have been developed. The Glass connects to the handheld via Bluetooth. For Bluetooth connection between Glass and the handheld, the MyGlass application is installed and set up on the handheld device. MyGlass is needed for pairing and connecting to the Glass from the handheld via Bluetooth.

4.2.1 GlassHandheldActivity

GlassHandheldActivity is the main Android activity that runs on the handheld. In this activity, an instance of *BlueToothAdapter* is obtained via the *BlueToothManager* class in the Android Bluetooth API (Tang 2014). The *BlueToothAdapter* instance is used to check for Bluetooth support and execution of various Bluetooth tasks. *GlassHandheldActivity* implements the *DisseminationServiceProxyListener* interface of DisService, thus having access to a proxy to DisService. In the *OnCreate* method, an asynchronous task, *initializeDisServiceTask*, is executed. In this task, a proxy to DisService is instantiated and initialized. The proxy subscribes to the wearable group in DisService. *GlassHandheldActivity* is then registered as a listener to the

subscribed proxy. This enables *GlassHandheldActivity* to send and receive notifications from other wearables subscribed to the wearable group in DisService. The *dataArrived* method of the *DisseminationServiceProxyListener* interface is a callback method that is invoked when new data arrive via DisService. The data are of type “byte array” and are converted to a string.

The *sendMessageToGlass* method is then invoked using the received data as an argument. Inside this method, the *BlueToothAdapter* instance is used to invoke the *getBondedDevices* method. This method returns a list of *BluetoothDevice* objects that are currently or previously paired with the handheld device. A *BluetoothDevice* class represents a remote Bluetooth device that helps in establishing connection to the device and query for information on the Bluetooth device. The *BluetoothDevice* list is then traversed to identify the Glass Bluetooth-enabled wearable device. Once the Glass wearable is identified, the *ConnectToGlass* class, which extends the *Thread* class is instantiated. This class is responsible for providing a connection to the given Bluetooth device and sending the message obtained by the handheld application via DisService to the connected Bluetooth device.

The constructor of the *ConnectToGlass* class takes the paired Glass wearable device (*BlueToothDevice*) and the string message as arguments. The *BlueToothDevice* object is then used to invoke the *createRfcommSocketToServiceRecord* method. This method takes a universally unique identifier (UUID) as an argument. The UUID is a service record UUID used to lookup the RFCOMM channel device. This method returns RFCOMM *BlueToothSocket* ready for an outgoing connection. RFCOMM is a connection-oriented, streaming transport over Bluetooth. Using *BluetoothSocket* as the argument, the connect method is invoked, which attempts to connect to the Glass wearable. This is a blocking call until a connection is established or the connection attempt fails. If the connection is unsuccessful, the exception message is logged and the socket is closed. If the connection is successful, the *writeToGlass* method is invoked and the connected socket is passed as an argument. The connected socket is then used to open the input/output (IO) streams. The *getOutputStream* method is invoked to obtain an *OutputStream* object. The data obtained by the application via DisService are written to the *OutputStream* object to deliver the message to the Glass wearable via the connected *BluetoothSocket*. Following this, the *OutputStream* and *BluetoothSocket* are closed. Finally, the *RunOnUIThread* method is invoked to update the main user interface indicating that the message has been sent to the Glass wearable.

4.2.2 GlassActivity

GlassActivity is the main Android activity that runs on the Glass wearable. This activity is responsible for listening to messages from the handheld paired to the Glass wearable via Bluetooth. In the *OnCreate* method of *GlassActivity*, the *setContentView* method is invoked to facilitate placement of UI elements in the Glass's activity. Following this, an instance of *BlueToothAdapter* via the *BlueToothManager* is obtained for executing the Bluetooth tasks necessary to connect to the handheld device. The *BlueToothAdapter* instance is used to check if Bluetooth is supported and enabled in the Glass. If the check yields a positive result, an object of a private class, *ListenThread*, is instantiated anonymously. *ListenThread* is a private class that extends the *Thread* class. It is mainly responsible for creating and managing *BlueToothServerSocket* to listen for incoming Bluetooth connection requests.

In the constructor of *ListenThread*, the *BlueToothAdapter* instance is used to invoke the *listenUsingRfcommWithServiceRecord* method and is passed as a UUID. The UUID is similar to that used on the client side so that it can recognize and accept the client's incoming connection request. This method returns *BlueToothServerSocket*, which keeps listening for an incoming client connection request or until it encounters an exception. Once a client connection is established, *BlueToothServerSocket* returns *BluetoothSocket*. Following this, close method is invoked on *BlueToothServerSocket*, which closes *BlueToothServerSocket*, as it is not required any further. Following this, a new thread, *ReadThread*, is spawned and *BluetoothSocket* is passed to it as a parameter. This thread is responsible managing the connected socket and transferring data between the handheld (client) and Glass (server).

In the constructor of *ReadThread*, the IO streams are opened by an invocation of the *getInputStream* method, which returns an *InputStream* object. The data from the *InputStream* object are read incrementally in a while loop. Once the entire data set is read, Glass's UI is notified of data retrieval by invoking the *runOnUIthread* method. An explicit intent is created to help launch a new Android activity, *DisplayAnimation*, for displaying the message retrieved by *GlassActivity*. The message is added to the intent by invoking the *putExtra* method on the newly instantiated intent object. The *DisplayAnimation* activity is launched via the *startActivity* method and passing the intent as an argument to the *startActivity* method.

4.2.3 DisplayGestureActivity

DisplayAnimationActivity is an Android activity responsible for displaying animations on Glass to convey the message received by *GlassActivity* from the handheld device. In the *OnCreate* method, the *setContentView* method is called for the placement of views within the activity window. The intent object is retrieved by invocation of the *getIntent* method on the current activity. On retrieval of Intent object, the *getStringExtra* method is called on the intent object by passing the string key as a parameter. The method returns the string value of the key stored in the intent object. Once the message is retrieved, an appropriate image is selected from the resources folder to load *ImageView* objects within the activity window space. Property animations is then applied to the *ImageView* object, which changes the property of *ImageView* over time thus animating the views.

4.3 Smartwatch Application

The handheld application and the smartwatch wearable application has been developed using the Android Wear API. The Android Wear package is one of the packages in the Google Play Services Package. For Android Wear development, the Google Play Services SDK is added to the Android Studio Project, which gives access to the Android Wear API. The Android Wear operating system that runs on the wearables uses the Bluetooth link to connect to the handheld device. On the handheld device, the Android Wear Play application is installed and set up to pair and connect to the wearable smartwatch device.

4.3.1 SmartwatchHandheldActivity

SmartwatchHandheldActivity is the main Android activity that runs on the handheld device. It implements the *DisseminationServiceProxyListener* interface of DisService. It also implements the *GoogleAPIClient* interfaces, *ConnectionCallbacks* and *OnConnectionFailedListener*, in the Wear API. The Google API client provides an entry point to the Google Play Services and manages the network connection between the service and the device itself. The *OnCreate* method is invoked to initialize the activity.

The *setContentView* method is invoked to facilitate placement of UI elements in *SmartwatchHandheldActivity*. An instance of *GoogleApiClient* using the *GoogleApiClient.Builder* APIs is created inside the *OnCreate* method. The method *addApi* is invoked on the instance of *GoogleApiClient* to add the Wear API to *GoogleApiClient*. The connect method of *GoogleApiClient* is used to connect to the Google Wear API in the Google Play Services Library. The

ConnectionCallbacks and *OnConnectionFailedListener* interfaces receive callbacks as a response to the asynchronous connect method of the *GoogleApiAclient* instance.

Lastly, in the *OnCreate* method, an asynchronous task, *initializeDisServiceTask*, is executed. In this task, a proxy to DisService is instantiated and initialized. The proxy subscribes to the wearable group in DisService. *SmartwatchHandheldActivity* is then registered as a listener to the subscribed proxy. This enables *SmartwatchHandheldActivity* to send and receive notifications from other wearables subscribed to the wearable group in DisService. The *dataArrived* method of the *DisseminationServiceProxyListener* interface is a callback method that is invoked when new data as a byte array arrive via DisService. The *dataArrived* method converts the byte array into a string format and notifies the main UI thread of the arrival of the string message. Following this, the method *pushDataToWearable* is invoked with the received data as an argument. This method is responsible for syncing the message from the handheld to the wearable smartwatch using the Wearable Data API. The API provides access to the data layer of a data communications link between the handheld and the wearable. In the *pushDataToWearable* method, the *isConnected* method is invoked on the *GoogleApiClient* instance, which basically checks for connection to the Google Wear API in the Google Play Services Library.

If the connection exists, an instance of *PutDataMapRequest* is created. It is a helper class that helps in the creation of a *DataMap* object. *DataMap* encapsulates the data that get exchanged over the wearable data layer. It can contain a collection of data types stored as key/value pairs. A string path is specified to uniquely identify the *DataMap* object that gets created via *PutDataMapRequest.* The method *GetDataMap* is invoked on the *PutDataMapRequest* object to obtain a *DataMap* object. The *DataMap* object is then set with the current time and the message received by *SmartwatchHandheldActivity* via DisService. The method *asPutDataRequest* is invoked on the instance of *PutDataMapRequest* to obtain a *PutDataRequest* object, which is used to create new data items in the Android Wear network. Following this, the *putDataItem* method in the Data API is invoked to add the data items in the Android Wear network. The method takes the *GoogleAPiClient* instance and the *PutDataRequest* object as its parameters. It returns *PendingResult* as a result of calling an API method in the Google Play Services. The status of the operation is retrieved via *PendingResult*. This way the handheld adds the data items into the Android Wear network, which can be received by the wearable on the other end via a Bluetooth link.

4.3.2 WearService

On the wearable smartwatch, an Android-based service, *WearService*, runs in the background and monitors the Wear data layer for any new data items in the Android Wear network. In order to listen for data layer events, *WearService* extends the abstract *WearableListenerService* class and implements the *onDataChanged* method. This method gets called when data items are created, changed, or deleted on the Wear network. The argument to the *onDataChanged* method is an instance of *DataEventBuffer*, which is a data structure that holds the references to a set of *DataEvent* instances in the Wear Data layer. Each of the *DataEvent* instances in the set is processed to retrieve a *DataItem* object. Each *DataItem* object is converted to a *DataMapItem* object, which is used to retrieve the *DataMap* instance by invoking the *getDataMap* method on the *DataMapItem* instance. The method *getString* is invoked on the *DataMap* instance. It takes a key of type "string" as an argument. The key used in the argument is the same key used in the handheld application to set the *DataMap* value. If the key matches the key contained in the current *DataMap* instance, it returns the value of the key in the current *DataMap* instance.

The value is the actual data that were sent by the handheld to wearable via the Wear network using the Wear API. An explicit intent is created to help launch a new activity, *NotificationActivity*, for displaying the data retrieved by *WearService* running on the smartwatch. The data are added to the intent by invoking the *putExtra* method on the newly instantiated intent object. *NotificationActivity* is launched via the *startActivity* method by passing the intent as an argument to the *startActivity* method.

4.3.3 DisplayGestureActivity

NotificationActivity is an Android activity responsible for displaying the data received by *WearService* from the handheld device. The *OnCreate* method is invoked to initialize *NotificationActivity*. The method *setContentView* is invoked to facilitate placement of UI elements in *NotificationActivity*. Following this, the *getIntent* method is invoked to retrieve the intent that started *NotificationActivity*. The method *getStringExtra* is invoked with the name of the extra data as an argument. The name of the extra data is the same name that was used in *WearService* to add extended data to the intent. The method returns the extra data value associated with the name. This value is then set in a *TextView* object to be displayed within the *NotificationActivity* window.

5. Conclusion

This research presents an architecture for and illustrates the feasibility of implementing wearable information sensing, dissemination, and retrieval to inform context in a PCE such as those used by many Army tactical applications. This work documents an approach to developing such a solution using commercial-off-the-shelf wearable devices and sensors, shown in the operation of an integrated system. The prototype framework demonstrated that Army-specific gestures can be recognized, dynamically disseminated, and appropriately modulated for retrieval thus reinforcing the invisibility and localized scalability concepts intrinsic in true PCEs. While in this effort only the feasibility of such an architecture and implementation is demonstrated, future experiments can be performed using the developed framework to evaluate the efficacy of presentation adaption given contextualized cognitive or kinetic state information, task or mission-level goals, and virtual-human teaming mixes. Future work is also planned to investigate the quantitative value of this information to intelligently prioritize and filter information flows in broader scenario contexts, such as tactical and command and control decision making.

6. Notes

1. android.bluetooth, 15 Sept 2015. Developers, Android [accessed 2015]. http://developer.android.com/reference/android/bluetooth/package-summary.html.
2. Myo Android SDK, 2014. Thalmic Labs [accessed 2015]. https://developer.thalmic.com/docs/api_reference/android/index.html.

7. References

Suri N, Benincasa G, Tortonesi M, Stefanelli C, Kovach J, Winkler R, Kohler R, Hanna J, Pochet L, Watson SC. Peer-to-peer communications for tactical environments: observations, requirements, and experiences. In IEEE Communications Magazine. October 2010;48(10):60–69.

Suri N, Benincasa G, Choy S, Formaggi S, Gilioli M, Interlandi M, Kovach J, Rota S, Winkler R. Disservice: A peer-to-peer disruption tolerant dissemination service. Military Communications Conference. 2009. MILCOM 2009. IEEE

Tang Jeff. Beginning Google Glass Development. Apress. Chapter 7, Networking, Bluetooth, and Social, 2014.

Weiser M. The Computer for the 21st Century. Scientific American (265:3) 1991, pp 66–75.

Satyanarayanan M. Pervasive Computing: Vision and Challenges. IEEE Personal Communications (8:44) 2001, pp 10-17

Weinstein E, Ho P, Heisele B, Poggio T, Steele K, Agarwal A. Handheld face identification technology in a pervasive computing environment. Short Paper Proceedings of International Conference on Pervasive Computing (Pervasive 2002), Zurich, Switzerland, 2002.

Chen G, Kotz D. Solar: A pervasive computing infrastructure for context-aware mobile applications. Department of Computer Science, Dartmouth College, Hanover, NH, USA.

Chen W, Jiang Z, Wu Z. AnyCom: A component framework optimization for pervasive computing. In Lecture Notes in Computer Science, Springer Berlin/Heidelberg, 2005, pp. 236–242.

Plymale WO. Pervasive Computing Goes to School. EDUCAUSE Review (40:1) 2005, pp 60–61.

Want R, Schilit BN, Adams NI, Gold R, Petersen K, Goldberg D, Ellis JR, Weiser M. The ParcTab ubiquitous computing experiment. Xerox Palo Alto Research Center, Palo Alto, CA, USA.

Narendra NC, Umesh B, Nandy SK, Kalapriya K. Functional and architectural adaptation in pervasive computing environments. In Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing, ACM Press, Grenoble, France, 2005.

Cherniack M, Franklin MJ, Zdonik S. Data management for pervasive computing. A tutorial given at the 27th International Conference on Very Large Data Bases (VLDB 2001), Rome, Italy, 2001.

List of Symbols, Abbreviations, and Acronyms

API	application programming interface
DIL	disconnected, intermittent, and limited
GDK	Glass Development Kit
IDE	integrated development environment
IO	input/output
NIST	National Institute of Standards and Technology
PCE	pervasive computing environment
SDK	software development kit
UI	user interface
UUID	universally unique identifier

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

2 (PDF) DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT

1 (PDF) GOVT PRINTG OFC A MALHOTRA

5 (PDF) DIRECTOR US ARMY RESEARCH LAB RDRL CII B BROOME RDRL CII B SOMIYA METU LAUREL SADLER ROBERT WINKLER STEPHEN RUSSELL